SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

Department of Computer Science
California Institute of Technology

Caltech-CS-TR-91-03

1 March 1991

Reporting Period:	1 October 1990 — 28 February 1991
Principal Investigator:	Charles L. Seitz
Faculty Investigators:	Alain J. Martin
	Charles L. Seitz
	Jan L. A. van de Snepscheut

Sponsored by the
Defense Advanced Research Projects Agency
DARPA Order Number 6202

Monitored by the
Office of Naval Research
Contract Number N00014-87-K-0745

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
01 MAR 1991		**01-10-1990 to 28-02-1991**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Submicron Systems Architecture	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**16**	

SUBMICRON SYSTEMS ARCHITECTURE

Department of Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This document is a summary of research activities and results for the five-month period, 1 October 1990 to 28 February 1991, under the Defense Advanced Research Project Agency (DARPA) Submicron Systems Architecture Project. Previous semiannual technical reports and other technical reports covering parts of the project in detail are listed following these summaries, and can be ordered from the Caltech Computer Science Library.

1.2 Objectives

The central theme of this research is the architecture and design of VLSI systems appropriate to a microcircuit technology scaled to submicron feature sizes. Our work is focused on VLSI architecture experiments that involve the design, construction, programming, and use of experimental multicomputers (message-passing concurrent computers), and includes related efforts in concurrent computation and VLSI design.

2. Architecture Experiments

2.1 Mosaic Project

Chuck Seitz, Nanette J. Boden, Jakov Seizovic, Wen-King Su

Mosaic C is an experimental *fine-grain multicomputer* aimed at exploring a region of the space of multicomputer designs that is shifted by two orders of magnitude from today's medium-grain systems. Whereas today's medium-grain multicomputers typically employ hundreds of nodes with megabytes of storage per node, a Mosaic system may employ tens of thousands of nodes with tens of kilobytes per node.

The attraction of the fine-grain form of multicomputer is, first of all, that it offers greater performance/cost than the medium-grain form. A specific advantage of the Mosaic C design is that a node with only tens of kilobytes of storage can be implemented with a single VLSI chip. The single-chip Mosaic C node (see the plot on the following page) includes 64KB of primary storage; read-only storage for initialization, self-test, and bootstrap; a 16-bit processor and packet interface; and a high-performance, asynchronous, 2D-mesh router.

We are currently starting the construction of a 16K-node Mosaic system based on these single-chip nodes. Mosaic program-development systems are, however, already in routine use for programming-system and application development.

The Mosaic C project includes numerous interacting subtasks ranging from chip design and system packaging to programming-system development and application studies. Progress in these tasks is described below. Of particular interest, however, is that over the past five months a contract was negotiated with Hewlett-Packard Company for the non-recurring engineering, packaging tooling, and first set of prototype 64-node circuit boards; this contract is now signed, and prototype boards are expected to be completed in July.

During the past five months, work on programming systems for fine-grain multicomputers has proceeded in a coordinated way on several levels, including:

1. An examination of the programming notation we employ for expressing message-passing programs. These studies have revealed a relatively easy way in which we can express message-passing programs in our standard semantics using a subset of C++. We are just finishing the implementation of a prototype of this system, and will report in greater detail on this aspect of our work in the next semiannual technical report.

2. Implementation of programming tools, *eg,* compilers, loaders, *etc.* The results of some of these efforts are described below; an additional related effort is described in section 3.2.

3. The implementation of an advanced runtime system for the Mosaic. The results of these efforts are described below; related research on runtime-system design fundamentals is described in section 3.1.

Plot of the Mosaic C node.

Two expositions of multicomputer architecture, design, programming systems, and applications, including many details of the Mosaic design and programming system, were published during the past five months:

Charles L. Seitz, "Concurrent Architectures," Chapter one in *VLSI and Parallel Computation,* edited by Robert Suaya and Graham Birtwistle, Morgan Kaufmann Publishers, 1990.

Charles L. Seitz, "Multicomputers," Chapter five in *Developments in Concurrency and Communication,* edited by C. A. R. Hoare, Addison-Wesley, 1991.

2.1.1 The Memoryless Mosaic Chip

The Memoryless Mosaic MM3.0 chip was our first attempt to incorporate the asynchronous router into the Mosaic framework. As indicated in our previous semiannual technical report, testing of the MM3.0 chip discovered two minor design errors. The corrected version of the Memoryless Mosaic was submitted for fabrication in September 1990 as MM3.1. The chips arrived in the beginning of November, were subjected to nearly exhaustive tests, and proved to be fully functional. The yield on this run was $46/50 = 92\%$. These MM3.1 chips have been used extensively in our program-development boards.

With the MM3.1 chips now functional, continued refinement of the Memoryless Mosaic chips has been aimed at performance improvements and easier integration into the program-development boards.

Our performance target for the single-chip Mosaic nodes at $V_{dd} = 4$V is 40MHz operation for the synchronous parts and 80MB/s (an equivalent word rate) for the asynchronous router channels. Tests of the 64KB Mosaic memory show that it operates correctly up to 44MHz, and the router and packet interface to 88MB/s. In spite of the off-chip path to memory, the processor in the MM3.1 was able to operate up to 40MHz except for a small number of critical paths that are revealed in certain instructions between 30MHz and 40MHz operation.

A series of communication-intensive tests revealed a critical path that included the static PLA used for the conditional-branch instruction. A reexamination of timing requirements confirmed that it was possible instead to use our standard, precharged NOR-NOR PLA. Under the new timing, the condition-code PLA operates in domino-logic style with the clock phases reversed from those of the microcode PLA.

The MM3.1 tests also discovered that one of the y channels on the program-development board had its bit-order reversed. This problem was not discovered with MM3.0 because it was masked off by the timing-margin problems described in our previous report.

MM3.2 was submitted for fabrication on 5 December 1990 with the improved condition-code PLA and the y-channel bit order reversed. (Although the bit-reversal

problem was, strictly speaking, a problem with the circuit board, it was easier to fix it on the chip.) The only other change from MM3.1 was to delay the driving of address pins. In the original design, address drivers were a bit faster then we expected, causing the precharged value on the internal address bus to make a sharp spike on the address output pins as well. This spike unnecessarily increases the $\frac{dI}{dt}$ noise, so the delays were changed to eliminate it.

The MM3.2 chip was returned from fabrication on 14 February 1991, and proved to be fully functional.

The run on which the MM3.2 was fabricated was of significantly lower quality than the usual MOSIS 1.2μm runs. Yield was only $39/53 = 74\%$, where our usual experience for MM3 chips is close to 90%. As usual, we tried to identify every single fault to assure that there is no marginal circuitry or layout in the design. Six of the 14 bad chips were damaged during the bonding phase, having mechanical damage around some of the bonding pads, and/or missing or shorted bonding wires. Fabrication defects in four more chips have been positively identified. The remaining 4 chips contain probable, but invisible, fabrication defects.

The devices on these MM3.2 chips were also at least 10% slower than previous runs, including the MM3.1 chips. This phenomenon had a fortunate side-effect of enabling us to trace additional critical paths that would probably have been masked by the the speed limit of our program-development boards.

An MM3.3 chip was submitted for fabrication on February 19; this contains a few adjustments to the driver sizes, and attempts improvement on the detected critical paths. We plan to populate one program-development board with faster, 15ns SRAM chips, to enable us to test the chips at the design target frequency of 40MHz.

With all of the silicon parts now thoroughly tested, we have assembled the layout of the full Mosaic C node. (See the plot on page 3.) This chip is approximately 9.25mm$\times$10.25mm in 1.2μm MOSIS SCMOS design rules.

2.1.2 Mosaic C Self-Test and Bootstrap Program

A version of the ROM-resident program for the full Mosaic has been created using the new C compiler. The program performs a number of self-test routines, waits for a message to arrive, and branches to the program contained in the message body. The self-test routines include: register test, ALU test, memory test, and packet-interface test, in that order. The register test propagates a shifting bit pattern through the processor's 24 data registers. The ALU test performs a fixed set of operations on a set of predefined number pairs. The memory test rotates and shifts a 33-word bit pattern through the memory. To combine the packet interface test with memory tests, the bit patterns are written through the packet interface from the processor to itself. The reading and writing of the pattern accomplishes tests of parts of the router, the synchronizer, the fifos, and the message DMA mechanism.

2.1.3 Manufacturing Contract

A contract has been negotiated and now signed with Hewlett-Packard Company to generate photomasks, fabricate a first run of chips on their CMOS34 line, develop test programs, design and build the TAB-packaging tooling, and assemble and test three prototype 8×8 Mosaic boards. The goal of this effort is prepare for manufacturing 8×8 boards in quantity. The target cost for these boards in larger lots is less than $5000 (less than $78/node(!)).

2.1.4 Mosaic C Compiler

The Mosaic C compiler is an adaptation of the gnu C compiler. The compiler has been greatly improved since our previous report. It now accepts the full C language with the exception of floating-point data types, and it produces highly optimized assembly code.

The compiler itself is only a part of the tool set that one would need for compiling a C program into runable Mosaic programs. The compiler converts C programs into assembly-language programs. Additional tools are needed to convert assembly-language programs into object-code files, to combine object-code files, and to maintain compiler libraries. Instead of spending our efforts reinventing the wheel, we borrowed the Sun-4 compiler tool set for use with Mosaic programs. By defining the Mosaic object-code format and library format to be identical to those of the Sun-4, we can use the Sun-4 `ld` and `ar` programs on Mosaic object-code files and libraries directly.

Furthermore, we spared ourselves the full detail of Sun-4 object-code format and the work that is required to write a full assembler. A simple pre-assember is used to translate a mosaic assembly-language file into an intermediate token file containing a sequence of generic assembler directives (.word, .text, .global, etc), labels, and constants. The partially digested token file is then converted by the Sun-4 assembler into Sun-4 object files.

(This section is the DARPA-required demonstration that we try to "work smart, not hard.")

2.1.5 Reactive Kernel for Mosaic Program-Development Boards

The targeting of the gnu C compiler for the Mosaic has enabled us to port the Reactive Kernel node operating system to the Mosaic. During the last four months, a number of both computation- and communication-intensive test programs have been written in this environment.

2.1.6 Runtime System Implementation

Work has been progressing on the implementation of fine-grain multicomputer runtime systems that distribute memory demands throughout the nodes of the

machine so that no one node overflows its memory bounds until a significant fraction of the total memory of the machine has been consumed. This implementation depends strongly on the selective-receive mechanism described in previous reports. A prototype runtime system including these algorithms was developed in C and debugged on a simulator that runs on medium-grain multicomputers.

With the completion of the Mosaic C compiler, this prototype runtime system was compiled for the Mosaic. We are currently in the process of porting this runtime system to the Mosaic program-development boards for debugging and profiling. As part of this porting process, we have defined rudimentary machine-loading procedures and host interfaces, as well as defining the minimal bootstrapping support required for the runtime system.

2.1.7 Copyless Message Handling

In our previous report, we mentioned that two Mosaic runtime system prototypes had been developed in C, each with a different approach to local node memory management. Since that time, we have fully adopted the principles of one of those approaches. This approach can be best termed as a *copyless* runtime system. One of the innate properties of distributed-memory architectures is the amount of copying that is performed, for example, in message passing where data is copied from the memory of one node to the memory of another node. The goal of this copyless approach is to limit copying to message passing; no copying should occur within the local node memory. We have practically achieved this goal in the current prototype by utilizing the broad flexibility of the Mosaic message-passing system. (It is interesting that for Mosaic and advanced medium-grain systems such as the Symult S2010, copying by message passing from the memory of one node to the memory of another is actually *faster* than copying within local memory.)

When a packet arrives at a node, its contents are written into a section of Mosaic memory delimited by two pointers set by the runtime system. If the whole message fits into the provided buffer, a *message-received* interrupt is set for the runtime system. If the entire message does not fit, a *buffer-full* interrupt is set. In this case, the remainder of the packet is blocked into the message network, and a new buffer must be provided. The contents received thus far, if needed later, should be copied by the runtime system into the new buffer and then the pointers reset so that the remaining message contents are written into the new buffer. Clearly, to avoid copying one must use buffer sizes that are large enough to accommodate the incoming message, but to consistently use buffer sizes that are larger than the incoming message simply wastes memory.

Using the remarkably flexible interrupt mechanisms of Mosaic, we can implement a two-phase receive for messages to ensure that all buffers for incoming messages are of the correct size. We first set the pointers for the incoming message to a small buffer that is the size of a message header, the first word of which is the length of the incoming message (minus the header). As the message arrives,

the header is written into the small buffer. When that buffer is full, the *buffer-full* interrupt is set. The runtime system extracts the length information from the buffer and allocates a new buffer of that size to receive the remainder of the message. When the pointers to the new message buffer have been set, the remainder of the message is written directly into the new buffer. The header of the message need not be copied, and this buffer can be manipulated directly throughout the life-cycle of the message.

The practicality of this technique depends on the relative speeds of the network and of the buffer allocation algorithm. While the allocation routine is finding a new buffer of the correct size for the incoming message, other messages are blocked in the network. Preliminary analysis indicates that this technique will not significantly degrade the overall performance of the machine in spite of this blocking, because the packet interface and routers include a substantial amount of buffering.

2.2 Second-Generation Medium-Grain Multicomputers*

Chuck Seitz, Joe Beckenbach, Christopher Lee, Jakov Seizovic, Craig Steele, Wen-King Su

Our Caltech project continues to work with the DARPA-supported Touchstone project at the Intel Supercomputer Systems Division, principally in connection with the mesh-routing chips for the Delta prototype (see section 4.4). The acquisition of a 570-node Delta system by a Caltech-led consortium was announced in November. The routing-mesh backplane for this 30Gflop system is 16 nodes high by 36 nodes wide, and is based on the FMRC2.3 mesh-routing chip.

The project currently operates the following medium-grain multicomputers: 8-node and 64-node Cosmic Cubes, a 128-node Intel iPSC/1, a 16-node Intel iPSC/2, and 32-node and 192-node Symult S2010 systems. The 192-node S2010 system, which is the machine most preferred by users, is accessed through the Caltech Concurrent Supercomputer Facilities. Utilization of this system continues to be at a level of approximately 90% of the available node-hours.

The project continues to distribute the Cosmic Environment package to several additional sites each month.

* This segment of our research is sponsored jointly by DARPA and by a grant from Intel Supercompter Systems Division (Beaverton, Oregon).

3. Concurrent Computation

3.1 Fine-Grain-Multicomputer Runtime Systems

Nanette J. Boden, Chuck Seitz

These studies of the design of distributed runtime systems for fine-grain multicomputers are closely coordinated with the implementation of the experimental runtime systems on the Mosaic, described in section 2.1.6 and 2.1.7.

In our previous report, we listed several undue restrictions commonly found in medium-grain multicomputer programming. These restrictions are related to assumptions adopted to simplify the implementation of the runtime systems. For example, a multicomputer program may continue to execute so long as any node that has been chosen to receive a new process can allocate memory resources to accommodate the code and data for the process. When some node runs out of memory, however, the computation will halt (with an "out of memory" indication).

For medium-grain machines, which typically have megabytes of primary storage per node, this failure point occurs only after some significant fraction of the total memory of the machine has been consumed. If the storage of one node of a ten-node machine is exhausted, then at least one-tenth of the memory is in use. More importantly, if the storage load was even approximately balanced, much more of the total memory of the machine was actually in use before the failure.

In contrast, the memory of one node of a fine-grain multicomputer represents a tiny fraction of the total memory of the machine. The typically tens of kilobytes of storage contained in a fine-grain multicomputer node precludes treating the node as having boundless storage. If we did not take active measures to distribute and balance the storage demands of a computation, a small fluctuation in the memory requirements of a node could cause the computation to fail, even though the great majority of the total memory in the machine might not be in use. In addition, we should not consider the nodes as possessing all of the storage available. Thus, a fine-grain multicomputer might be more usefully modeled as an ensemble of nodes with finite storage plus at least one node with infinite storage. In practice, the nodes with infinite storage are those that host disks. This infinite aspect of the abstract machine is required so that we can reason about computations whose storage demands are unbounded.

Instead of basing the design of the runtime system of a fine-grain multicomputer on assumptions that are clearly not reasonable for this architecture, we have been developing distributed algorithms for system functions such as process creation and code distribution. These algorithms do not rely on the presence of boundless storage at every node. Most of the algorithms we have devised employ the selective-receive mechanism discussed in our previous report. The selective-receive mechanism is implemented using only reactive semantics and process creation.

These runtime-system algorithms include a user-process creation mechanism that permits a congested node to refuse a process-creation request. The availability of a selective receive mechanism also provides opportunities for experimention with sparse distribution of process code. We are currently investigating various alternatives for dispersing and later executing process code. The experimental vehicle for this work is the Mosaic runtime system described in section 2.1.6.

3.2 A Pascal Compiler for the Mosaic

Jan L. A. van de Snepscheut, Johan J. Lukkien

Experiments with different implementations and semantics of communication primitives have favored a version in which it is not necessary to maintain a receive queue of incoming messages. The ability to write various implementations in Pascal instead of assembly language contributed significantly to the number and scope of these experiments. Our favorite version has now been incorporated into the compiler, and is the only change we have made to it during the past six months. The performance that we get from this Pascal implementation is quite good even though no sophisticated code optimizations are performed.

For example, register allocation is done in a crude manner. For the dhrystone benchmark (on one processor) we measure a performance of 2494 cycles per iteration, which on the present Mosaic program-development system (with a 25MHz clock) delivers a speed of 10,000 dhrystones/sec. At the projected clock speed of 40 MHz this implies 16,000 dhrystones/sec.

3.3 Fluid-Flow Computations

Jan L. A. van de Snepscheut, H. Peter Hofstee

The Mosaic program-development systems have been used to carry out some fluid-flow computations. The model that we use is based on equilibrium flux flow, a direct simulation method for compressible, inviscid, ideal-gas flow, which is well-suited to conditions for high-speed aerodynamics. The present program uses a fixed, regular grid. The granularity of the computation depends on the granularity of the grid.

This approach lends itself well to parallel computation, and we measure good speedups, but the fixed grid limits its applicability. We are now working on an algorithm in which the shape and granularity of the grid are adapted by the computation. This obvious idea turns out to require a not-so-obvious review of the interpretation of the grid.

3.4 Formal Methods for Concurrency

Jan L. A. van de Snepscheut, Johan J. Lukkien

Large circuits are constructed from smaller circuits, and in doing so we prefer to use the functional description of the building blocks rather than having to look

inside the smaller circuits and figure out their operation in the context of the large circuit. This "black box" approach requires some discipline from the designer of the building blocks. For software, one would like to follow the same pattern, but in this case no design discipline has yet been established. Although programs in which we are interested only in their initial and final states can be handled satisfactorily, no compositional methods for reasoning about the intermediate states of programs exist. This is vital, however, for understanding reactive and concurrent programs, and is the key to any form of stepwise refinement. We have encouraging results in dealing with intermediate states, as described in Johan Lukkien's PhD thesis, but we have not yet been able to handle concurrency.

Attempts to give a precise specifcation of the sliding-window protocol has uncovered an error in one of the versions of the algorithm under the usual weak assumptions on communication channels. We have shown that the error does not occur in a slightly different (and more efficient) version, or if one makes stronger assumptions about the channel.

3.5 Distributed Resource Management

Jan L. A. van de Snepscheut, H. Peter Hofstee, Johan J. Lukkien

We have written a number of widely different implementations of a functional programming language. The language has no notion of process or memory management and, therefore, serves as a test vehicle for seeing how well we can provide those automatically in an implementation. Distribution of computational activities (evaluating a function in a processor where time is available) is done automatically and we are comparing different versions, both by experiment and by theory. Automatic storage management (storing data in a processor where space is available) remains to be done. Presently, space is allocated in the simplest possible way, namely, in the processor in which the request for more storage originates.

3.6 A Concurrent Wire-Routing Program

Su-Lin Wu, Chuck Seitz

We are attempting to develop a highly concurrent program to generate wire routings for circuit boards and VLSI chips. As reported previously, this program adapts the Lee-Moore algorithm for finding the shortest path between two points to a method of finding good (low-cost) routes of n points. By taking advantage of existing electrically equivalent wires, this heuristic gives better routes than does simply applying the two-point algorithm repeatedly.

An initial prototype of this program has been completed. The prototype has exposed areas in which more concurrency might be achieved. We are now interfacing this program with a VLSI CAD tool to study and correct the inefficiencies discovered.

3.7 The Page Kernel

Craig S. Steele, Chuck Seitz

The previously described Page Kernel (PK) concurrent programming environment, operating on the Symult S2010 multicomputer, has continued to improve in performance and reliability. PK is an evolution of the now-familiar reactive programming model. High-performance message systems allow the visible distinctions between shared- and distributed-memory machines to be obscured, permitting the programmer to access shared data structures much as in a shared-memory machine. PK uses the virtual-memory capabilities of second-generation multicomputer nodes to implement data-sharing mechanisms supporting multiple, overlapping address spaces.

Some small optimizations have resulted in significant performance increases. Each multicomputer node has a software-maintained cache of data structures called *blocks* that have been accessed by *actions* (a light-weight, reactive process). Reference counting allows these cached copies to be purged from the cache if not in use. However, in computations that are actively creating new actions, it was found to be advantageous to change from an eager to a lazy algorithm that defers purging until either the cache is full or no actions are scheduled for execution. This change and an analogous decision to defer reclamation of action virtual context resources can particularly affect the startup speed of computations and the overall speed of small, highly concurrent problems, for which initial costs can be significant.

Actions are coded as C++ functions, and have actual arguments bound to the formal parameter list upon instantiation. The arguments may include portable (between actions) references to globally accessible data structures. Converting the portable form of the reference to a localized pointer within an action's address space is a relatively expensive operation. Caching the localized pointer for subsequent use has dramatically sped up some calculations.

With these and several other incremental improvements, PK programs demonstrate high efficiency even on one or two multicomputer nodes. For example, the performance penalty of a 128×128 matrix-multiply example run on a single node with a concurrent formulation using 128 actions is currently 41% when compared with a sequential version.

4. VLSI Design

4.1 CAD for Asynchronous Circuit Synthesis

Dražen Borković, Alain J. Martin

We are pursuing our effort towards the realization of a fully integrated CAD system for the design of asynchronous VLSI circuits. Ultimately, the system will integrate synthesis, placement and routing, and performance-analysis and circuit-optimization tools.

The heart of the synthesis tool is a program (PRGEN) that generates *minimal* production rules from handshaking expansion of the program to be compiled into a circuit. Previously, PRGEN could automatically compile only straightline programs. It has now been extended to handle choice (IF-statements). Once nested IF-statements have been added, it will be possible to mechanically compile all programs into optimal production rules. (We can already automatically generate sized-transistor layouts from production rules.)

The program allows the designer to provide additional information in order to obtain optimal circuits. The information that corresponds to the knowledge of the high-level program can be provided in a structured way that reflects the organization of the specification. Since not all information is known at the high level (initial stage of the design), assertions can be made about lower-level properties of the design.

4.2 Asynchronous Circuit Design in Gallium Arsenide

José A. Tierno, Alain J. Martin

The GaAs circuits designed in the previous reporting period were received from fabrication and tested for functionality and speed. Most of the subcircuits were found to be working correctly; speed was greatly reduced by a design flaw in the pad driver circuit, which made internal delays very difficult to measure. Another test circuit was designed and sent for fabrication with new pad designs, some test circuitry and a fast asynchronous incrementer.

A timing model was developed that provides fast evaluations of circuit performance. This model was incorporated into an event-driven simulation program (called PRSIM) that uses the timing information generated by the model from a transistor description of the circuit. PRSIM was originally developed for CMOS, and was adapted recently to GaAs. The delay model is accurate to within 10% of a VSPICE simulation, Vitesse's version of the spice2g program.

Several other CMOS tools were modified for GaAs, especially our standard-cell place-and-route program. A full set of cells was designed to work with this program, and these are being used in the current microprocessor project.

We have started to design a GaAs implementation of the Caltech Asynchronous Microprocessor. Most of the datapath cells have already been designed and are

being laid out. The control logic will be generated automatically. We are waiting for the results of the last-submitted chip to get feedback on the pad design and support circuitry.

Unfortunately, at the moment, the lack of good switch-level simulation tools for the type of GaAS circuits we are using makes us quite vulnerable to trivial errors during hand layout or hand compilation.

4.3 Testing Asynchronous Circuits

Pieter J. Hazewindus, Alain J. Martin

We have shown that a single stuck-at fault in a non-redundant asynchronous circuit results in a transition either not taking place or firing prematurely, or both, during an execution of the circuit. This result contradicts a widespread belief that asynchronous circuits are "self-testing" because a single stuck-at fault always results in the circuit halting.

A transition not taking place can be tested easily, as this always prevents a transition on a primary output from taking place. A premature firing can also be tested but the addition of testing points may be required to enforce the premature firing and to propagate the transition to a primary output. Hence all single stuck-at faults are testable. All test sequences can be generated from the high-level specification of the circuit. The circuits are hazard-free in normal operation and during the tests.

A simple scan design is used to connect the testing points. The design is an asynchronous adaptation of a shift-register queue.

The conclusion of this preliminary investigation of asynchronous circuit testing can be put in the "bad-news/good-news" form. The bad news is that, contrary to common belief, a single stuck-at fault does not always lead to the circuit halting, but may also lead to some transitions firing prematurely. The good news is that, also contrary to common belief, any single stuck-at fault can be tested.

We are redesigning the control part of the Caltech Asynchronous Microprocessor so as to make the circuit fully testable. A first redesign indicates that the number of extra testing points is very small. Three of the five processes that constitute the control of the microprocessor each require one testing point. An extra testing point is required when the processes are connected together. We have not yet investigated how the test sequences should be generated. They all can be generated from the high-level specification of the circuit, but that may be too expensive—in terms of the size of the vectors—for certain circuits, *eg,* datapaths. We have also ignored some complications created by the reset procedures.

4.4 Fast, Self-Timed, Routing and Communication Chips

Chuck Seitz, Jim Miller

We are working with a group of students from the Caltech VLSI-design class to develop a new set of routing automata from which a wide variety of routing and communication chips will be assembled. These new routing automata employ two-cycle signaling internally, and are expected to be easier to compose than the cells in the Frontier-series mesh-routing chips (FMRC chips).

The first test chip is a *slack chip* that relaxes non-interference protocols to allow high-bandwidth communication over cables up to about 20m in length. This chip is expected to be submitted for fabrication in 1.2μm MOSIS SCMOS within the next month, and, if successful, will be used to connect between Mosaic program-development boards and arrays.